Square Always Exponentiation

Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, Vincent Verneuil

Square Always Exponentiation

Christophe Clavier[1], Benoit Feix[1,2], Georges Gagnerot[1,2], Mylène Roussellet[2], and Vincent Verneuil[2,3]

[1] XLIM-CNRS, Université de Limoges, France
firstname.familyname@unilim.fr
[2] INSIDE Secure, Aix-en-Provence, France
firstname-first-letterfamilyname@insidefr.com
[3] Institut de Mathématiques de Bordeaux, Talence, France

Abstract. Embedded exponentiation techniques have become a key concern for security and efficiency in hardware devices using public key cryptography. An exponentiation is basically a sequence of multiplications and squarings, but this sequence may reveal exponent bits to an attacker on an unprotected implementation. Although this subject has been covered for years, we present in this paper new exponentiation algorithms based on trading multiplications for squarings. Our method circumvents attacks aimed at distinguishing squarings from multiplications at a lower cost than previous techniques. Last but not least, we present new algorithms using two parallel squaring blocks which provide the fastest exponentiation to our knowledge.

Keywords: Public key cryptography, exponentiation, long integer arithmetic, side-channel analysis, atomicity.

1 Introduction

Nowadays most embedded devices implementing public key cryptography use RSA [16] for encryption and signature schemes, or cryptographic primitives over $(\mathbb{F}_p, \times)$ such as DSA [7] and the Diffie-Hellman key agreement protocol [6]. All these algorithms require the computation of modular exponentiations. Since the emergence of the so-called *side-channel analysis*, embedded devices implementing these cryptographic algorithms must be protected against a wider and wider class of attacks.

Moreover, the cost and timing constraints are crucial in many applications of embedded devices (e.g. banking, transport, etc.). This often requires cryptographic implementors to choose the best compromise between security and speed. Improving the efficiency of algorithms or countermeasures generates thus a lot of interest in the industry.

An exponentiation is generally processed using a sequence of multiplications, some of them having different operands and some of them being squarings. In [2], Amiel et al. showed that this distinction can provide exploitable side-channel

leakages to an attacker. Classical countermeasures consist of using exponentiation algorithms where the sequence of multiplications and squarings does not depend on the secret exponent.

Our contribution is to propose a new exponentiation scheme using squarings only, which is faster than the classical countermeasures. Also, we introduce new algorithms having a particularly low cost when two squarings can be parallelized.

This paper is organized as follow: in Section 2 we recall classical exponentiation algorithms and present some well-known side-channel attacks and countermeasures. Then we propose our new countermeasure in Section 3 and study its efficiency from the parallelization point of view in Section 4. Finally we present some practical results in Section 5 and we conclude in Section 6.

2 Background on Exponentiation on Embedded Devices

We recall in this section some classical exponentiation algorithms. First we present the *square-and-multiply* algorithms upon which are based most of the exponentiation methods. Then we introduce the *side-channel analysis* and in particular the *simple power analysis* (SPA). We present some algorithms immune to this attack, and we finally recall a particular side-channel attack aimed at distinguishing squarings from multiplications in an exponentiation operation.

2.1 Square-and-Multiply Algorithms

Many exponentiation algorithms have been proposed in the literature. Among the numerous references an interested reader can refer for instance to [14] for details. Alg. 2.1 and Alg. 2.2 are two variants of the classical square-and-multiply algorithm which is the simplest approach to compute an RSA exponentiation.

Alg. 2.1 Left-to-Right Square-and-Multiply Exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$
Output: $m^d \mod n$

1: $a \leftarrow 1$
2: **for** $i = k-1$ **to** 0 **do**
3: $\quad a \leftarrow a^2 \mod n$
4: $\quad$ **if** $d_i = 1$ **then**
5: $\quad\quad a \leftarrow a \times m \mod n$
6: **return** a

Considering a balanced exponent d, these algorithms require on average $1S + 0.5M$ per bit of exponent to perform the exponentiation – S being the cost of a modular squaring and M the cost of a modular multiplication. It is generally considered in the literature – and corroborated by our experiments – that on cryptographic coprocessors $S \approx 0.8M$.

These algorithms are no longer used in embedded devices for security applications since the emergence of the side-channel analysis.

Alg. 2.2 Right-to-Left Square-and-Multiply Exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \mod n$
1: $a \leftarrow 1$; $b \leftarrow m$
2: **for** $i = 0$ **to** $k - 1$ **do**
3: **if** $d_i = 1$ **then**
4: $a \leftarrow a \times b \mod n$
5: $b \leftarrow b^2 \mod n$
6: **return** a

2.2 Side-Channel Analysis on Exponentiation

Side-channel analysis was introduced in 1996 by Kocher in [12] and completed in [13]. Many attacks have been derived in the following years.

On one hand, *passive* attacks rely on the following physical property: a microprocessor is physically made of thousands of logical gates switching differently depending on the executed operations and on the manipulated data. Therefore the power consumption and the electromagnetic radiation, which depend on those gates switchings, reflect and may leak information on the executed instructions and the manipulated data. Consequently, by monitoring such side-channels of a device performing cryptographic operations, an observer may infer information on the implementation of the program executed and on the – potentially secret – data involved.

On the other hand, *active* attacks intend to physically tamper with computations and/or stored values in memories. Such effects are generally obtained using clock or power glitches, laser beam, etc.

Finally some works [1] have highlighted the fact that passive and active attacks may be combined to threaten implementations applying countermeasures against both of them but not against their simultaneous use.

In the remainder of this section we focus on two passive attacks : the SPA presented hereafter with classical countermeasures, and a particular analysis from [2] discussed in Section 2.3.

Simple Power Analysis Simple side-channel analysis [11] consists in observing a difference of behavior depending on the value of the secret key on the component performing cryptographic operations by using a single measurement.

In the case of an exponentiation, the original SPA is based on the fact that, if the squaring operation has a different pattern than a multiplication, the secret exponent can be directly read on the curve. For instance, in Alg. 2.1, a 0 exponent bit implies a squaring to be followed by another squaring, while a 1 bit causes a multiplication to follow a squaring. Classical countermeasures consist of using *regular* algorithms or applying the *atomicity* principle, as detailed in the following.

Alg. 2.3 Montgomery Ladder Exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \mod n$
1: $R_0 \leftarrow 1$; $R_1 \leftarrow m$
2: **for** $i = k - 1$ **to** 0 **do**
3: $\quad R_{1-d_i} \leftarrow R_0 \times R_1 \mod n$
4: $\quad R_{d_i} \leftarrow {R_{d_i}}^2 \mod n$
5: **return** R_0

Alg. 2.4 Left-to-Right Multiply Always Exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \mod n$
1: $R_0 \leftarrow 1$; $R_1 \leftarrow m$; $i \leftarrow k - 1$; $t \leftarrow 0$
2: **while** $i \geq 0$ **do**
3: $\quad R_0 \leftarrow R_0 \times R_t \mod n$
4: $\quad t \leftarrow t \oplus d_i$; $i \leftarrow i - 1 + t$ [$\oplus$ is bitwise XOR]
5: **return** R_0

Regular Algorithms These algorithms include the well known *square-and-multiply always* and Montgomery ladder algorithms [15,10]. The latter is presented hereafter in Alg. 2.3. It is generally preferred over the square-and-multiply always method since it does not involves dummy multiplications which makes it naturally immune to the C safe-error attacks [18,10].

Such regular algorithms perform one squaring and one multiplication at every iteration and thus require $1M + 1S$ per exponent bit.

Atomicity Principle This method, presented by Chevallier-Mames et al. in [4], can be applied to protect the square-and-multiply algorithm against the SPA. It yields the so-called *multiply always* algorithm, since all squarings are performed as classical multiplications. We present a left-to-right multiply always algorithm in Alg. 2.4.

The interest of the multiply always algorithm is its better performances compared to the regular ones. Indeed it performs an exponentiation using on average $1.5M$ per exponent bit.

2.3 Distinguishing Squarings from Multiplications

Amiel et al. showed in [2] that the average Hamming weight of the output of a multiplication $x \times y$ has a different distribution whether:

- the operation is a squaring performed using the multiplication routine, i.e. $x = y$, x uniformly distributed in $[0, 2^k - 1]$,
- or the operation is an "actual" multiplication, i.e. x and y independent and uniformly distributed in $[0, 2^k - 1]$.

This attack can thus target an atomic implementation such as Alg. 2.4 where the same multiplication operation is used to perform $x \times x$ and $x \times y$.

First, many exponentiation curves using a fixed exponent but variable data have to be acquired and averaged. Then, considering the average curve, the aim of the attack is to reveal if two consecutive operations are identical – i.e. two squarings – or different – i.e. a squaring and a multiplication. As in the classical SPA, two consecutive squarings reveal that a 0 bit has been manipulated whereas a squaring followed by a multiplication reveals a 1 bit. This information is obtained using the above-mentioned leakage by subtracting the parts of the average curve corresponding to two consecutive operations: peaks occur if one is a squaring and the other is a multiplication while subtracting two squarings should produce only noise. It is worth noticing that no particular knowledge on the underlying hardware implementation is needed which in practice increases the strength of this analysis.

A classical countermeasure against this attack is the randomization of the exponent[4], i.e. $d^* \leftarrow d + r\varphi(n)$, r being a random value. The result is obtained as $m^d \bmod n = m^{d^*} \bmod n$.

In spite of the possibility to apply the exponent randomization, this attack brings into light an intrinsic flaw of the multiply always algorithm: the fact that at some instant a multiplication performs a squaring ($x \times x$) or not ($x \times y$) depending on the exponent. In the rest of this paper we propose new atomic algorithms that are exempt from this weakness.

3 Square Always Countermeasure

We present in this section new exponentiation algorithms which simultaneously benefit from efficiency of the atomicity principle and immunity against the aforementioned weakness of the multiply always method.

3.1 Principle

It is well known that a multiplication can be performed using squarings only. Therefore we propose the following countermeasure which consists in using either expression (1) or (2) to perform all the multiplications in the exponentiation. Combined with the atomicity principle, this countermeasure completely prevents the attack described in Section 2.3 since only squarings are performed.

$$x \times y = \frac{(x+y)^2 - x^2 - y^2}{2} \tag{1}$$

$$x \times y = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2 \tag{2}$$

[4] Notice however that the randomization of the message has no effect on this attack, or even makes it easier by providing the required data variability.

At the first glance, (1) requires three squarings to perform a multiplication whereas (2) requires only two. Further analysis reveals however that using (1) or (2) in Alg. 2.1 and 2.2 has always the cost of replacing multiplications by twice more squarings. Indeed, notice that in the multiplication $a \leftarrow a \times m$ of Alg. 2.1 m is a constant operand. Therefore implementing $a \times m$ using (1) yields $y = m$, thus $m^2 \mod n$ can be computed only once at the beginning of the exponentiation. The cost of computing y^2 can then be neglected.

This trick does not apply to Alg. 2.2 since no operand is constant in step 4. However $b \leftarrow b^2$ is the following operation. Using equation (1) in Alg. 2.2 then yields to store $t \leftarrow y^2$ and save the following squaring: $b \leftarrow t$. The resulting cost is thus equivalent as trading one multiplication for two squarings.

Remark In our context, (1) or (2) refer to operations modulo n. Notice however that divisions by 2 in these equations require neither inversion nor multiplication. For example, we recommend computing $z/2 \mod n$ in the following atomic way:

$t_0 \leftarrow z$
$t_1 \leftarrow z + n$
$\alpha \leftarrow z \mod 2$
return $t_\alpha/2$

3.2 Atomic Algorithms

Trading multiplications for squarings in Alg. 2.1 and 2.2 just requires to apply formula (1) or (2) at step 5 in Alg. 2.1 or step 4 in Alg. 2.2. However the resulting algorithms would still present a leakage since different operations would be performed when processing a 0 or 1 bit. Hence it is necessary to apply the atomicity principle on these algorithms.

This step is achieved by identifying a minimal pattern of operations to be performed on each loop iteration and rewrite the algorithms using this pattern. For the considered algorithms, the minimal pattern should obviously contain a single squaring since it is the only operation required by the processing of a 0 bit and performing dummy squarings would lessen the performances of the algorithm. An addition, subtraction and division by 2 should also be present to compute formulas (1) or (2). Finally some more operations are required to manage the loop counter and the pointer on exponent bits.

Algorithm 3.1 presented hereafter details how to implement atomically the square always method in a left-to-right exponentiation using equation (1).

As in [4] we use a matrix for a more readable and efficient implementation:

$$M = \begin{pmatrix} 1\,1\,1\,0\,2\,1\,1\,1\,2\,1 \\ 2\,0\,1\,2\,2\,2\,2\,2\,3\,0 \\ 1\,1\,3\,0\,0\,0\,0\,2\,0\,0 \\ 3\,3\,3\,0\,3\,3\,1\,1\,3\,1 \end{pmatrix}$$

The main loop of Alg. 3.1 can be viewed as a four state machine where each row j of M define the operands of the atomic pattern. The atomic pattern itself is

Alg. 3.1 Left-to-Right Square Always Exponentiation with (1)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \mod n$
1: $R_0 \leftarrow 1$; $R_1 \leftarrow m$; $R_2 \leftarrow 1$; $R_3 \leftarrow m^2/2 \mod n$
2: $j \leftarrow 0$; $i \leftarrow k-1$
3: **while** $i \geq 0$ **do**
4: $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_{M_{j,2}} \mod n$
5: $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \mod n$
6: $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}}/2 \mod n$
7: $\quad R_{M_{j,6}} \leftarrow R_{M_{j,7}} - R_{M_{j,8}} \mod n$
8: $\quad j \leftarrow d_i(1 + (j \mod 3))$
9: $\quad i \leftarrow i - M_{j,9}$
10: **return** R_0

given by the content of the loop, i.e. steps 4 to 9. An exponent bit d_i is processed by the state $j = 0$ (resp. $j = 3$) if the previous bit d_{i+1} is a 0 (resp. a 1). This state is followed by the processing of the next bit if $d_i = 0$, or by the states $j = 1$ and $j = 2$ if $d_i = 1$. For more clarity, we present below the four sequences of operations corresponding to each state. The dummy operations are identified by a $\star$.

$j = 0$ ($d_i = 0$ or 1)	
$R_1 \leftarrow R_1 + R_1 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$R_2 \leftarrow R_1/2 \mod n$	$\star$
$R_1 \leftarrow R_1 - R_2 \mod n$	$\star$
$j \leftarrow d_i$	[$\star$ if $d_i = 0$]
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

$j = 2$ ($d_i = 1$)	
$R_1 \leftarrow R_1 + R_3 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$R_0 \leftarrow R_0/2 \mod n$	
$R_0 \leftarrow R_2 - R_0 \mod n$	
$j \leftarrow 3$	
$i \leftarrow i - 1$	

$j = 1$ ($d_i = 1$)	
$R_2 \leftarrow R_0 + R_1 \mod n$	
$R_2 \leftarrow {R_2}^2 \mod n$	
$R_2 \leftarrow R_2/2 \mod n$	
$R_2 \leftarrow R_2 - R_3 \mod n$	
$j \leftarrow 2$	
$i \leftarrow i$	$\star$

$j = 3$ ($d_i = 0$ or 1)	
$R_3 \leftarrow R_3 + R_3 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$R_3 \leftarrow R_3/2 \mod n$	$\star$
$R_1 \leftarrow R_1 - R_3 \mod n$	$\star$
$j \leftarrow d_i$	
$i \leftarrow i - (1 - d_i)$	[$\star$ if $d_i = 1$]

We also present in Alg. 3.2 a right-to-left variant of the square always exponentiation using equation (2). This algorithm requires the following matrix:

$$M = \begin{pmatrix} 0\,0\,2\,0\,0\,0\,2\,1 \\ 2\,1\,2\,2\,1\,0\,1\,0 \\ 0\,2\,1\,1\,0\,0\,2\,0 \\ 0\,0\,0\,0\,1\,2\,1\,1 \end{pmatrix}$$

Alg. 3.2 Right-to-Left Square Always Exponentiation with (2)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$
Output: $m^d \mod n$
1: $R_0 \leftarrow m$; $R_1 \leftarrow 1$; $R_2 \leftarrow 1$
2: $i \leftarrow 0$; $j \leftarrow 0$
3: **while** $i \leq k-1$ **do**
4: $\quad j \leftarrow d_i(1 + (j \mod 3))$
5: $\quad R_{M_{j,0}} \leftarrow R_{M_{j,1}} + R_0 \mod n$
6: $\quad R_{M_{j,2}} \leftarrow R_{M_{j,3}}/2 \mod n$
7: $\quad R_{M_{j,4}} \leftarrow R_{M_{j,5}} - R_{M_{j,6}} \mod n$
8: $\quad R_{M_{j,3}} \leftarrow {R_{M_{j,3}}}^2 \mod n$
9: $\quad i \leftarrow i + M_{j,7}$
10: **return** R_1

As for the previous algorithm, the main loop of Alg. 3.2 has four states. Here, the state $j = 0$ corresponds to the processing a 0 bit and the sequence $j = 1$, $j = 2$, and $j = 3$ corresponds to the processing of a 1 bit, as detailed below.

$j = 0$ ($d_i = 0$)	
$j \leftarrow 0$	[$\star$ if j was 0]
$R_0 \leftarrow R_0 + R_0 \mod n$	$\star$
$R_2 \leftarrow R_0/2 \mod n$	$\star$
$R_0 \leftarrow R_0 - R_2 \mod n$	$\star$
$R_0 \leftarrow {R_0}^2 \mod n$	
$i \leftarrow i + 1$	

$j = 2$ ($d_i = 1$)	
$j \leftarrow 2$	
$R_0 \leftarrow R_2 + R_0 \mod n$	$\star$
$R_1 \leftarrow R_1/2 \mod n$	
$R_0 \leftarrow R_0 - R_2 \mod n$	$\star$
$R_1 \leftarrow {R_1}^2 \mod n$	
$i \leftarrow i$	$\star$

$j = 1$ ($d_i = 1$)	
$j \leftarrow 1$	
$R_2 \leftarrow R_1 + R_0 \mod n$	
$R_2 \leftarrow R_2/2 \mod n$	
$R_1 \leftarrow R_0 - R_1 \mod n$	
$R_2 \leftarrow {R_2}^2 \mod n$	
$i \leftarrow i$	$\star$

$j = 3$ ($d_i = 1$)	
$j \leftarrow 3$	
$R_0 \leftarrow R_0 + R_0 \mod n$	$\star$
$R_0 \leftarrow R_0/2 \mod n$	$\star$
$R_1 \leftarrow R_2 - R_1 \mod n$	
$R_0 \leftarrow {R_0}^2 \mod n$	
$i \leftarrow i + 1$	

3.3 Performance Analysis

Algorithms 3.1 and 3.2 are mostly equivalent in terms of operations realized in a single loop. The number of dummy operations (additions, subtractions and halvings) introduced to fill the atomic blocks are the same in the two versions – it is generally considered that the cost of these operations is negligible compared to multiplications and squarings. Both algorithms require $2S$ per exponent bit on average or $1.6M$ if $S/M = 0.8$ which represents a theoretical 11.1% speed-up over Alg. 2.3 which is the fastest known regular algorithm immune to the attack from [2]. Table 1 compares the efficiency of the multiply always, Montgomery ladder, and square always algorithms when $S = M$ and $S/M = 0.8$.

In addition, our algorithms can be enhanced using the sliding window or m-ary exponentiation techniques [14,9] while the Montgomery ladder cannot. These techniques are known to provide a substantial speed-up on Alg. 2.4 when extra memory is available. Though we did not investigate this path, we believe that a comparable trade-off between space and time can be expected.

Table 1. Comparison of the expected cost of SPA protected exponentiation algorithms (including the multiply always which is not immune to the attack from [2])

Algorithm	General cost	$S/M = 1$	$S/M = 0.8$	# registers
Multiply always (Alg. 2.4)	$1.5M$	$1.5M$	$1.5M$	2
Montgomery ladder (Alg. 2.3)	$1M + 1S$	$2M$	$1.8M$	2
L.-to-r. Square always (Alg. 3.1)	$2S$	$2M$	$\mathbf{1.6M}$	4
R.-to-l. Square always (Alg. 3.2)	$2S$	$2M$	$\mathbf{1.6M}$	3

3.4 Security Considerations

Our algorithms are protected against the SPA by the implementation of the atomicity principle. The analysis from [2] cannot apply either since only squarings are involved. As a matter of comparison, notice that the exponent blinding countermeasure does not fundamentally remove the source of the leakage but only renders this attack practically infeasible. Embedded implementations should also be protected against the *differential power analysis* (DPA) which we do not detail in this study. However it is worth noticing that classical DPA countermeasures, like exponent or modulus randomization, can be applied as well. The interested reader may refer to [13,5].

We recommend implementing Alg. 3.2 instead of Alg. 3.1 since left-to-right algorithms are vulnerable to the chosen message SPA and *doubling attack* [8], and more subject to combined attacks [1]. Besides, Alg. 3.2 requires one less register than Alg. 3.1.

It is well-known that algorithms using dummy operations generally succumb to safe-error attacks. Immunity to C and M safe-errors can be easily obtained by applying the exponent randomization technique, which also prevent the DPA. Nevertheless, special care has been taken in our algorithms to ensure that inducing a fault in any of the dummy operations would produce an erroneous result. For instance, in the following sequence of dummy operations in Alg. 3.2 ($j = 0$), no operation can be tampered with without corrupting R_0 and thus the result of the exponentiation:

$$\begin{aligned} R_0 &\leftarrow R_0 + R_0 \mod n \\ R_2 &\leftarrow R_0/2 \mod n \\ R_0 &\leftarrow R_0 - R_2 \mod n \end{aligned}$$

Only operations $i \leftarrow i$ and $j \leftarrow 0$, appearing in some instances of Alg. 3.1 and 3.2 patterns, have not been protected for readability reasons. It is easy to fix these points: perform $i \leftarrow i \pm M_{j,\cdot} + \alpha$ instead of $i \leftarrow i \pm M_{j,\cdot}$ in Alg. 3.1 and 3.2 and add a step $i \leftarrow i - \alpha$ in the loop. The $j \leftarrow d_i(1 + \dots)$ operation should be protected in the same manner. In the end, our algorithms are immune to C safe-error attacks.

Further work may focus on implementing on our algorithms the *infective computation* strategy presented by Schmidt et al. in [17] in order to counterfeit the combined attacks.

4 Parallelization

It is well known that the Montgomery ladder algorithm is well suited for parallelization. It is thus natural to ask if the square always algorithms have the same property. For example the two squarings needed to perform a classical multiplication using equation (2) are independent and can therefore be performed simultaneously. The same strategy applies for equation (1).

We believe that the interest of this section extends beyond the scope of embedded systems. Nowadays most of computers are provided with several processors which enables using parallelized algorithms to speed-up computations.

4.1 Parallelized Algorithms

We noticed that right-to-left exponentiations are more suited for parallelization than their left-to-right counterpart since more operations are independent. For example in Alg. 2.2 one can first perform all squarings (step 5), store all values corresponding to a $d_i = 1$, and then perform the remaining multiplications. We present in Alg. 4.1 a right-to-left square always algorithm using (2) and two parallel squaring blocks (i.e. two 1-operand multipliers). For a better readability Alg. 4.1 is not atomic and two operations o_1 and o_2 performed simultaneously are denoted $o_1 \mid\mid o_2$.

Algorithm 4.2 is an atomic variant of Alg. 4.1. It requires two extra registers compared to the non atomic version and the following matrices:

$$M = \begin{pmatrix} 1\,1\,5\,6\,5\,5\,5\,0\,1 \\ 0\,6\,4\,3\,0\,1\,3\,1\,1 \\ 2\,5\,3\,1\,5\,5\,5\,0\,0 \\ 2\,5\,0\,6\,0\,1\,5\,0\,1 \end{pmatrix} \quad N = \begin{pmatrix} 1\,1\,0 \\ 5\,2\,2 \end{pmatrix}$$

It is possible to further enhance the efficiency of Alg. 4.1 if more memory is available by storing more free squarings when 1's sequences are processed. This observation yields Alg. 4.3 which allows the storage of $extramax$ simultaneous precomputed squarings using as many registers R_3, R_4, $\dots R_{extramax+2}$. Alg. 4.3 with $extramax = 1$ is thus equivalent to algorithms 4.1 and 4.2. Though Alg. 4.3 is not atomic for readability reasons and because of the difficulty to write an

Alg. 4.1 Right-to-Left Parallel Square Always Exponentiation with (2)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} \dots d_0)_2$, require 5 k-bit registers a, b, R_0, R_1, R_2
Output: $m^d \mod n$

1: $a \leftarrow 1$; $b \leftarrow m$; $extra \leftarrow 0$
2: **for** $i = 0$ **to** $k - 1$ **do**
3: **if** $d_i = 1$ **then**
4: **if** $extra = 0$ **then**
5: $R_0 \leftarrow (a - b)^2 \mod n$ || $R_1 \leftarrow b^2 \mod n$
6: $a \leftarrow (a + b)^2 \mod n$ || $R_2 \leftarrow {R_1}^2 \mod n$
7: $a \leftarrow (a - R_0)/4 \mod n$
8: $b \leftarrow R_1$
9: $R_1 \leftarrow R_2$
10: $extra \leftarrow 1$
11: **else**
12: $R_0 \leftarrow (a - b)^2 \mod n$ || $a \leftarrow (a + b)^2 \mod n$
13: $a \leftarrow (a - R_0)/4 \mod n$
14: $b \leftarrow R_1$
15: $extra \leftarrow 0$
16: **else**
17: **if** $extra = 0$ **then**
18: $b \leftarrow b^2 \mod n$
19: **else**
20: $b \leftarrow R_1$
21: $extra \leftarrow 0$
22: **return** a

Alg. 4.2 Right-to-Left Atomic Parallel Square Always Exp. with (2)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1} d_{k-2} \dots d_0)_2$, require 7 k-bit registers R_0 to R_6
Output: $m^d \mod n$

1: $R_0 \leftarrow 1$; $R_1 \leftarrow m$; $v \leftarrow (0, 0, 0)$; $u \leftarrow 1$ [v_0 is i and v_1 is $extra$ from Alg. 4.1]
2: **while** $v_0 \leq k - 1$ **do**
3: $j \leftarrow d_{v_0}(v_1 + u + 1)$
4: $R_5 \leftarrow (R_0 - R_1)/2 \mod n$
5: $R_6 \leftarrow (R_0 + R_1)/2 \mod n$
6: $R_{M_{j,0}} \leftarrow {R_{M_{j,1}}}^2 \mod n$ || $R_{M_{j,2}} \leftarrow {R_{M_{j,3}}}^2 \mod n$
7: $R_{M_{j,4}} \leftarrow R_0 - R_2 \mod n$
8: $R_{M_{j,5}} \leftarrow R_3$
9: $R_{M_{j,6}} \leftarrow R_4$
10: $v_1 \leftarrow M_{j,7}$
11: $u \leftarrow M_{j,8}$
12: $t \leftarrow 1 - v_1(1 - d_{v_0+1})$
13: $R_{N_{t,0}} \leftarrow R_3$
14: $v_{N_{t,1}} \leftarrow 0$
15: $v_{N_{t,2}} \leftarrow v_{N_{t,2}} + 1$
16: $v_0 \leftarrow v_0 + u$
17: **return** R_0

atomic algorithm depending on a variable (here *extramax*), it should be possible to write an atomic version for each *extramax* value in the same way than we processed with Alg. 4.1.

Alg. 4.3 Right-to-Left Generalized Parallel Square Always Exp. with (2)

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$, $extramax \in \mathbb{N}^*$, require $extramax+4$ k-bit registers a, R_0, R_1, $\dots R_{extramax+2}$
Output: $m^d \mod n$

1: $a \leftarrow 1$; $R_1 \leftarrow m$; $extra \leftarrow 0$
2: **for** $i = 0$ **to** $k-1$ **do**
3: **if** $d_i = 1$ **then**
4: **if** $extra < extramax$ **then**
5: $R_0 \leftarrow (a - R_1)^2 \mod n$ || $R_{extra+2} \leftarrow {R_{extra+1}}^2 \mod n$
6: $a \leftarrow (a + R_1)^2 \mod n$ || $R_{extra+3} \leftarrow {R_{extra+2}}^2 \mod n$
7: $a \leftarrow (a - R_0)/4 \mod n$
8: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
9: $extra \leftarrow extra+1$
10: **else**
11: $R_0 \leftarrow (a - R_1)^2 \mod n$ || $a \leftarrow (a + R_1)^2 \mod n$
12: $a \leftarrow (a - R_0)/4 \mod n$
13: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
14: $extra \leftarrow extra-1$
15: **else**
16: **if** $extra = 0$ **then**
17: $R_1 \leftarrow {R_1}^2 \mod n$
18: **else**
19: $(R_1, R_2, \dots R_{extramax+1}) \leftarrow (R_2, R_3, \dots R_{extramax+2})$
20: $extra \leftarrow extra-1$
21: **return** a

Remark Notice that multiple assignments of steps 8, 13, and 19 may be traded for a cheap index increment if registers R_1, R_2, $\dots$, $R_{extramax+2}$ are managed as a cyclic buffer.

4.2 Cost of Parallelized Algorithms

We demonstrate in Appendix A that, as the length of the exponent tends to infinity, the cost per exponent bit of Alg. 4.3 tends to:

$$\left(1 + \frac{1}{4\,extramax+2}\right) S$$

It yields a cost of $7S/6$ for Alg. 4.1, 4.2, and 4.3 with $extramax = 1$, $11S/10$ for $extramax = 2$, $15S/14$ for $extramax = 3$, etc. The difference between this limit and costs actually observed in our simulations is negligible for 1024-bit or longer exponents.

It is remarkable that if $S/M = 0.8$ these costs become respectively $0.93M$, $0.88M$, $0.86M$, etc. per exponent bit. We believe that such performances cannot be achieved by binary algorithms using two parallelized 2-operands multiplication blocks. Indeed at least k multiplications have to be performed sequentially in Alg. 2.1 and 2.2, which requires at least $1M$ per exponent bit. Moreover when *extramax* tends to infinity, the cost of Alg. 4.3 tends to $1S$, which we believe to be the optimal cost of an exponentiation algorithm based on the binary decomposition of the exponent since k squarings at least have to be performed sequentially.

Table 2 summarizes the theoretical cost of parallelized algorithms cited in this study.

Table 2. Comparison of the expected cost of parallelized exponentiation algorithms

Algorithm	General cost	$S/M = 1$	$S/M = 0.8$
Parallelized Montgomery ladder	$1M$	$1M$	$1M$
Alg. 4.1, Alg. 4.2, Alg. 4.3 with *extramax* $= 1$	$7S/6$	$1.17M$	$\mathbf{0.93M}$
Alg. 4.3 with *extramax* $= 2$	$11S/10$	$1.10M$	$\mathbf{0.88M}$
Alg. 4.3 with *extramax* $= 3$	$15S/14$	$1.07M$	$\mathbf{0.86M}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
Alg. 4.3 with *extramax* $\to \infty$	$1S$	$1M$	$\mathbf{0.8M}$

5 Practical Results

In this section, we briefly present practical implementation results of the non-parallelized square always algorithm. As discussed in Section 3.4 we focused the right-to-left version.

We implemented this algorithm and the Montgomery ladder on an Atmel AT90SC smart card chip. This component is provided with an 8-bit AVR core and the AdvX coprocessor dedicated to long integer arithmetic. We used the Barrett reduction [3] to implement modular arithmetic.

We present in Table 3 the memory (code and RAM) and timing figures obtained with the chip and the AdvX running at 30 MHz. The observed speed-up of the square always algorithm over the Montgomery ladder is 5% on average. This is less than the predicted 11% but the difference can be explained by the neglected operations of the atomic pattern. Keep in mind that such results highly depend on the considered device and its hardware capabilities.

We performed careful SPA on both implementations and observed no leakage on power traces.

Table 3. On chip comparison of the Montgomery ladder and square always algorithms

Algorithm	Key Length (b)	Code Size (B)	RAM used (B)	Timings (ms)
Montgomery ladder (Alg. 2.3)	512	360	128	30
	1024	360	256	200
	2048	360	512	1840
Square Always (Alg. 3.2)	512	510	192	28
	1024	510	384	190
	2048	510	768	1740

6 Conclusion

In this paper we show that trading multiplications for squarings in an exponentiation scheme together with the atomicity principle provides a new countermeasure against side-channel attacks aimed at distinguishing squarings from multiplications. Moreover, this countermeasure is intrinsically more secure against such analysis than the classical multiply always atomic algorithm with exponent blinding, and provides better performances and flexibility towards space/time trade-offs than regular algorithms such as the Montgomery ladder or the square-and-multiply always.

As a complementary work, we present new algorithms using two parallel squaring blocks, and show how to write them atomically. We point out that, as far as we know, it leads to the fastest results in terms of speed. On the hardware side, an interesting conclusion is that two parallel squaring blocks enable faster exponentiation algorithms than two parallel multiplication blocks.

We believe that these observations are of great interest for the embedded devices industry and for everyone looking for fast exponentiation.

Acknowledgments

The authors would like to thank Sean Commercial and Alexandre Venelli for their valuable comments and advice on this manuscript. We would also like to thank the anonymous reviewers of this paper for their fruitful comments and advice.

References

1. F. Amiel, B. Feix, L. Marcel, and K. Villegas. Passive and Active Combined Attacks. In *Workshop on Fault Detection and Tolerance in Cryptography - FDTC 2007*. IEEE Computer Society, 2007.
2. F. Amiel, B. Feix, M. Tunstall, C. Whelan, and W. P. Marnane. Distinguishing Multiplications from Squaring Operations. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography - SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2008.

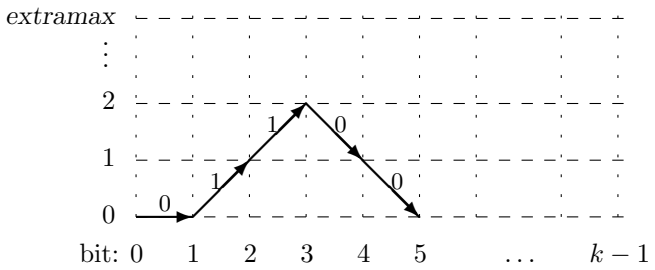
3. P. Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.
4. B. Chevallier-Mames, M. Ciet, and M. Joye. Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.
5. J-S Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 1999*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.
6. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
7. FIPS PUB 186-3. *Digital Signature Standard*. National Institute of Standards and Technology, october 2009.
8. P-A. Fouque and F. Valette. The Doubling Attack - *why upwards is better than downwards*. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 269–280. Springer, 2003.
9. D. Hankerson, A.J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Professional Computing Series, January 2003.
10. M. Joye and S-M. Yen. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 291–302, 2002.
11. P. Kocher, J. Jaffe, and B. Jun. Introduction to Differential Power Analysis and Related Attacks. 1998.
12. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
13. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
14. Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
15. P. L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *MC*, 48:243–264, 1987.
16. R. L. Rivest, A Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM 21*, pages 120–126, 1978.
17. Jörn-Marc Schmidt, Michael Tunstall, Roberto Maria Avanzi, Ilya Kizhvatov, Timo Kasper, and David Oswald. Combined implementation attack resistant exponentiation. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 305–322. Springer, 2010.
18. S-M. Yen and M. Joye. Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. *IEEE Trans. Computers*, 49(9):967–970, 2000.

A Cost of Algorithm 4.3

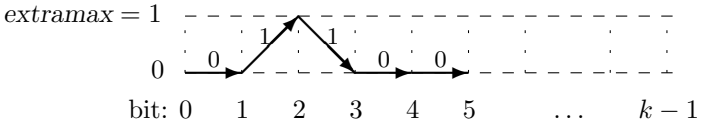
We present hereafter a demonstration of the claimed asymptotic cost of Alg. 4.3.

We first recall the principle of this algorithm: since 3 squarings are required to process a 1 bit, a fourth squaring slot is available at the same cost ($2S$). Thus, the algorithm scans the exponent from the right to the left and computes one squaring in advance at each 1 bit ($\nearrow$ in the following), within the limit of *extramax*. Then, as 0's are processed, the free squarings are consumed ($\searrow$) at null cost ($0S$). Two other cases may happen: first, a 1 bit can be processed but *extramax* squarings are already stored in registers, then one free squaring is consumed ($\searrow$) and $1S$ is enough to perform the two other squarings. Second, a 0 bit can be processed with no free squaring in registers ($extra = 0$). Only in this latter case one squaring is performed at the cost of $1S$ and the parallel squaring slot is wasted ($\rightarrow$).

We can consider the evolution of *extra* as exponent bits are processed using a diagram as below. For example, we have represented here the evolution of *extra* for the 5 first bits of an exponent $d = (d_{k-1} \dots 00110)_2$ with $extramax \geq 2$. The cost of the first 0 bit is $1S$ since $extra = 0$ at the beginning of the exponentiation, the cost of two next 1 bits is $2S$ each and *extra* is incremented, finally the two last 0 bits have cost $0S$ and *extra* is decremented. The total cost of the 5 bits is $5S$.



Observe now that the same bits have a higher cost if $extramax = 1$: as previously the two first bits 01 cost $1S$ and $2S$ respectively. However, the next 1 bit cannot lead to the computation of a second free squaring since $extramax = 1$. So the bit is processed at the cost of $1S$ and the free squaring is lost. Finally, the two last 0's cost $1S$ each since no free squaring is stored anymore. The cost of the sequence is $6S$.



For a given exponent and *extramax*, let's call a *c-cycle* a sequence of bits starting with $extra = c$, ending with $extra = c$, and inside which $extra > c$. In particular, we can decompose any exponent as a sequence of 0-cycles, except that the last one may be unterminated with $extra > 0$.

Then, let B_c^e stand for the expected number of bits of a c-cycle when $extramax = e$ and C_c^e its expected cost.

extramax = 1 For a random exponent and *extramax* = 1, a 0-cycle is "0" with probability 1/2 and "1x", $x \in \{0, 1\}$ otherwise. The cost of a 0-cycle "0" is $1S$ and the cost of a 0-cycle "1x" is $2S$ if $x = 0$ which happens with probability 1/2, or $3S$ if $x = 1$.

$$B_0^1 = 1/2 \times 1 + 1/2 \times 2 = 3/2$$

$$C_0^1 = 1/2 \times 1S + 1/2 \times (1/2 \times 2S + 1/2 \times 3S) = 7S/4$$

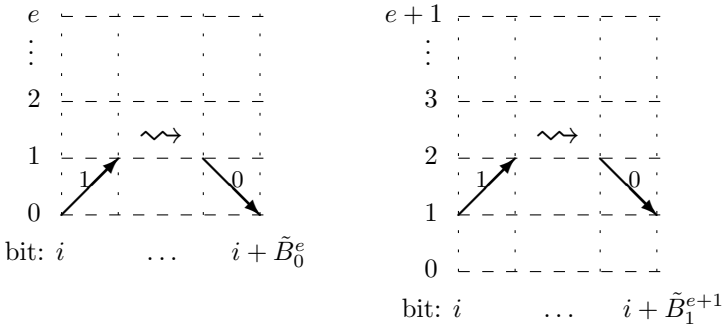
The expected cost of a 0-cycle with *extramax* = 1 is then $C_0^1/B_0^1 = 7S/6$ per bit. As the length of the exponent tends to infinity, the contribution of the possibly unterminated last 0-cycle becomes negligible. Therefore the cost per bit of a random exponent tends to the cost per bit of a 0-cycle as its length tends to infinity. So we can approximate the cost of algorithms 4.1, 4.2 and 4.3 to $7S/6$ for exponents of thousands of bits.

extramax = e A 0-cycle starts with a 0 with probability 1/2 and with a 1 otherwise. In the first case its cost is $1S$ as previously. Let $\tilde{B}_c^e$, respectively $\tilde{C}_c^e$, denote the expected length, respectively the expected cost, of a c-cycle starting with a 1 bit when *extramax* $= e$.

$$B_0^e = 1/2 \times 1 + 1/2 \times \tilde{B}_0^e \tag{3}$$

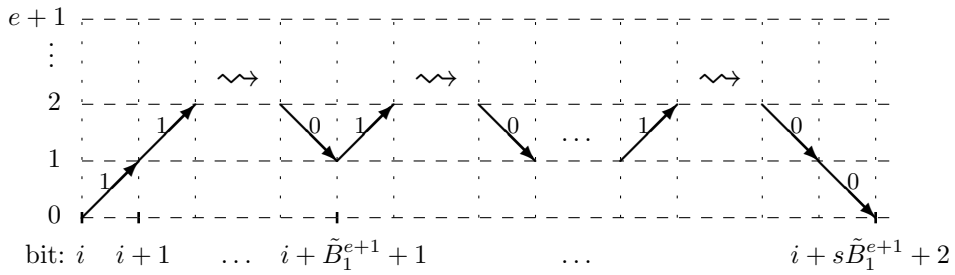
$$C_0^e = 1/2 \times 1S + 1/2 \times \tilde{C}_0^e \tag{4}$$

First we demonstrate that $\tilde{B}_0^e = 2e$. As depicted below, one can observe that $\tilde{B}_0^e = \tilde{B}_1^{e+1}$.



As depicted hereafter, the length $\tilde{B}_0^{e+1}$ of a 0-cycle with *extramax* $= e + 1$ and starting by a 1 bit is $s\tilde{B}_1^{e+1} + 2$ where s is the number of inner 1-cycles starting by a 1 bit. Notice also that $s = i$ with probability $2^{-(i+1)}$, which gives:

$$\tilde{B}_0^{e+1} = 2 + \sum_{i=0}^{\infty} \frac{i\tilde{B}_1^{e+1}}{2^{(i+1)}} = 2 + \tilde{B}_1^{e+1} = 2 + \tilde{B}_0^e$$



$(\tilde{B}_0^e)_{e\geq 1}$ is thus an arithmetic progression with common difference 2 and $\tilde{B}_0^1 = 2$. This yields $\tilde{B}_0^e = 2e$.

In a same manner, we can observe that:

$$\tilde{C}_0^{e+1} = 2S + \sum_{i=0}^{\infty} \frac{i\tilde{C}_1^{e+1}}{2^{(i+1)}} = 2S + \tilde{C}_1^{e+1} = 2S + \tilde{C}_0^e$$

Since $\tilde{C}_0^1 = 5S/2$ we obtain that $\tilde{C}_0^e = (1/2 + 2e)S$.

Using the above results in equations (3) and (4), we obtain finally:

$$B_0^e = 1/2 \times 1 + 1/2 \times 2e = 1/2 + e$$

$$\text{and} \quad C_0^e = 1/2 \times 1S + 1/2 \times (1/2 + 2e)S = (3/4 + e)S$$

The expectation of the cost per bit of a 0-cycle is then:

$$\frac{C_0^e}{B_0^e} = \left(\frac{3/4 + e}{1/2 + e}\right) S = \left(1 + \frac{1}{4e+2}\right) S$$

Therefore the expectation of the cost of Alg. 4.3 with *extramax* $= e$ tends then to $(1 + \frac{1}{4e+2})S$ as the length of the exponent tends to infinity.